


 devictoribero / **clean-code-javascript** Publicforked from [ryanmcdermott/clean-code-javascript](#)

Clean Code concepts adapted for JavaScript

 MIT license 830 stars  9.9k forks Star Watch ▾

Code

Issues


Pull requests 5

Actions

Projects

Security


Insights

 master ▾

...

This branch is [9 commits ahead](#), [62 commits behind](#) ryanmcdermott:master. #4

devictoribero ...

on Aug 9 [View code](#) README.md

clean-code-javascript

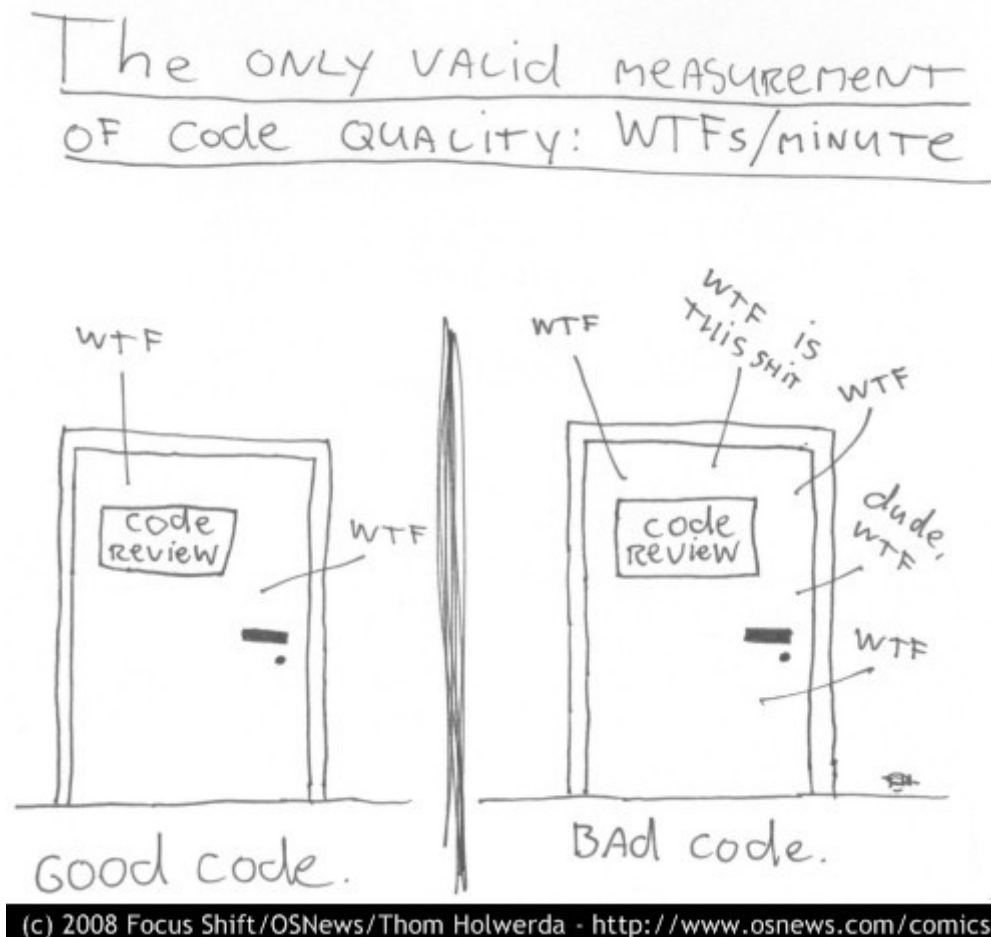
Este contenido no es original. está traducido de [aquí](#). Tampoco significa que todo lo que esté en este repositorio lo comparta. De hecho, hay unas cosas en las que no estoy de acuerdo.

Contenido

1. [Introducción](#)
2. [Variables](#)
3. [Funciones](#)
4. [Objetos y estructuras de datos](#)
5. [Clases](#)
6. [SOLID](#)
7. [Testing](#)

- 8. [Concurrencia](#)
- 9. [Manejo de errores](#)
- 10. [Formato](#)
- 11. [Comentarios](#)
- 12. [Traducciones](#)

Introducción



Principios de Ingeniería de Software por Robert C. Martin en el libro [Código Limpio](#), adaptado al Javascript. Esto no es una guía de estilos. Esto es una guía para crear código [legible, reutilizable y de fácil modificación](#) en Javascript.

No se deben seguir estrictamente todos los principios e incluso aún menos, como tampoco, éstos van a ser dogmas internacionales ni verdades absolutas. Los conceptos explicados no son más una compilación de buenas prácticas que han sido agrupadas a lo largo de muchos años de experiencia colectiva por los autores de *Código Limpio*.

Nuestro oficio de ingeniería de software tiene poco más de 50 años y todavía estamos aprendiendo mucho. Quizás cuando la arquitectura del software sea tan antigua como la arquitectura tradicional en sí misma, tengamos reglas más definidas que seguir. Por ahora, dejemos que estas pautas sirvan de faro para evaluar la calidad del código JavaScript que tu equipo y tu producís.

Una cosa más: Debes saber que estos principios, consejos o como quieras llamarlo, no te hará instantáneamente un mejor desarrollador de software y que trabajar con ellos durante muchos años tampoco significa que no vayas a hacer más errores. Cada trozo de código comienza como un primer borrador, igual que un jarrón precioso empieza con un trozo de arcilla feo y húmedo el cual vamos moldeando hasta conseguir el resultado final. Finalmente, limamos las imperfecciones cuando lo revisamos con nuestros compañeros a base de iteraciones. No te castigues por la posible mejora de los primeros borradores. En vez de eso, ¡Vence al código!

Variables

Utiliza nombres con sentido y de fácil pronunciación para las variables



Mal:

```
const yyyymmddstr = moment().format("YYYY/MM/DD");
```



Bien:

```
const fechaActual = moment().format("YYYY/MM/DD");
```



[Volver arriba](#)

Utiliza el mismo tipo de vocabulario para el mismo tipo de variables



Mal:

```
conseguirInformacionUsuario();  
conseguirDatosCliente();  
conseguirRegistroCliente();)
```



Bien:

```
conseguirUsuario();
```



[Volver arriba](#)

Utiliza nombres que puedan ser buscados

Leeremos más código del que jamás escribiremos. Es importante que el código que escribamos sea legible y se puede buscar en él. Al no crear variables que sean significativas para entender nuestro código... Estamos entorpeciendo a sus lectores. Haz tus variables sean fáciles de entender y buscar. Herramientas como [buddy.js](#) y [ESLint](#) pueden ayudar a identificar constantes no nombradas.

 **Mal:**

```
// Para que cojones sirve 86400000?  
setTimeout(blastOff, 86400000);
```

 **Bien:**

```
// Declaralas como constantes nombradas  
const MILISEGUNDOS_POR_DIA = 86400000;  
  
setTimeout(blastOff, MILISEGUNDOS_POR_DIA);
```

 [Volver arriba](#)

Utiliza variables explicativas

 **Mal:**

```
const direccion = "Calle Mallorca, Barcelona 95014";  
const expresionRegularCodigoPostalCiudad = /^[^,\s]+[,,\s]+(.*?)\s*(\d{5})?$/;  
guardarCP(  
  direccion.match(expresionRegularCodigoPostalCiudad)[1],  
  direccion.match(expresionRegularCodigoPostalCiudad)[2]  
);
```

 **Bien:**

```
const direccion = "One Infinite Loop, Cupertino 95014";  
const expresionRegularCodigoPostalCiudad = /^[^,\s]+[,,\s]+(.*?)\s*(\d{5})?$/;  
const [, ciudad, codigoPostal] =  
  direccion.match(expresionRegularCodigoPostalCiudad) || [];  
guardarCP(ciudad, codigoPostal);
```

 [Volver arriba](#)

Evita relaciones mentales

Explícito es mejor que implícito.

 **Mal:**

```
const ciudades = ["Barcelona", "Madrid", "Sitges"];
ciudades.forEach(l => {
  hacerAlgo();
  hacerAlgoMas();
  // ...
  // ...
  // ...
  // Espera, para que era `l`?
  dispatch(l);
});
```

 **Bien:**

```
const ciudades = ["Barcelona", "Madrid", "Sitges"];
ciudades.forEach(direccion => {
  hacerAlgo();
  hacerAlgoMas();
  // ...
  // ...
  // ...
  dispatch(direccion);
});
```

 **Volver arriba**

No añadas contexto innecesario

Si tu nombre de clase/objeto ya dice algo, no lo repitas en tu nombre de variable

 **Mal:**

```
const Coche = {
  marcaCoche: "Honda",
  modeloCoche: "Accord",
  colorCoche: "Azul"
};

function pintarCoche(coche) {
  coche.colorCoche = "Rojo";
}
```

 **Bien:**

```
const Coche = {
  marca: "Honda",
```

```
    modelo: "Accord",  
    color: "Azul"  
};  
  
function pintarCoche(coche) {  
    coche.color = "Rojo";  
}
```

 [Volver arriba](#)

Utiliza argumentos por defecto en vez de circuitos cortos o condicionales

Los argumentos por defecto suelen ser más limpios que los cortocircuitos. Ten en cuenta que si los usas, solo se asignará ese valor por defecto cuando el valor del parámetro sea `undefined`. Otros valores "falsos" como `''`, `" "`, `false`, `null`, `0` y `NaN`, no serán reemplazados por un valor predeterminado pues se consideran valores como tal.

 **Mal:**

```
function crearMicroCerveceria(nombre) {  
    const nombreMicroCerveceria = nombre || "Hipster Brew Co.";  
    // ...  
}
```

 **Bien:**

```
function crearMicroCerveceria(nombre = "Hipster Brew Co.") {  
    // ...  
}
```

 [Volver arriba](#)

Funciones

Argumentos de una función (idealmente 2 o menos)

Limitar la cantidad de parámetros de una función es increíblemente importante porque hacen que *las pruebas* de tu función sean más sencillas. Tener más de tres lleva a una locura combinatoria donde tienes que probar toneladas de casos diferentes con cada argumento por separado.

El caso ideal es usar uno o dos argumentos, tres... deben evitarse si es posible. Cualquier número superior a eso, debería ser agrupado. Por lo general, si tienes más de dos argumentos, tu función debe de estar haciendo demasiadas cosas. En los casos donde no es así, la mayoría de las veces un objeto de nivel superior será suficiente como un argumento *parámetro objeto*.

Ya que Javascript te permite crear objetos al vuelo sin tener que hacer mucho código repetitivo en una clase, puedes usar un objeto en caso de estar necesitando muchos parámetros.

Para indicar que propiedades espera la función, puedes usar las funcionalidades de desestructuración que nos ofrece ES2015/ES6. Éstas tienen algunas ventajas:

1. Cuando alguien mira la firma de la función, sabe inmediatamente que propiedades están siendo usadas
2. La desestructuración también clona los valores primitivos especificados del objeto argumento pasado a la función. Esto puede servir de ayuda para prevenir efectos adversos. *Nota: Los objetos y los arrays que son desestructurados del objeto parámetro NO son clonados.*
3. Las herramientas linter o *linterns* pueden avisarte de qué propiedades del objeto parámetro no están en uso. *Cosa que es imposible sin desestructuración.*

 **Mal:**

```
function crearMenu(titulo, cuerpo, textoDelBoton, cancelable) {  
  // ...  
}
```

 **Bien:**

```
function crearMenu({ titulo, cuerpo, textoDelBoton, cancelable }) {  
  // ...  
}  
  
crearMenu({  
  titulo: "Foo",  
  cuerpo: "Bar",  
  textoDelBoton: "Baz",  
  cancelable: true  
});
```

 **Volver arriba**

Las funciones deberían hacer una cosa

De lejos, es la regla más importante en la ingeniería del software. Cuando las funciones hacen más de una cosa, son difíciles de componer y *testear* entre otras cosas. Si isolamos las funciones por acciones, éstas pueden ser modificadas y mantenidas con mayor facilidad y tu código será mucho más limpio. De toda esta guía... si has de aprender algo, que sea esto. Ya estarás mmuy por delante de muchos desarrolladores de software.

 **Mal:**

```
function enviarCorreoAClientes(clientes) {  
  clientes.forEach(cliente => {  
    const historicoDelCliente = baseDatos.buscar(cliente);  
    if (historicoDelCliente.estaActivo()) {  
      enviarEmail(cliente);  
    }  
  });  
}
```

 **Bien:**

```
function enviarCorreoClientesActivos(clientes) {  
  clientes.filter(esClienteActivo).forEach(enviarEmail);  
}  
  
function esClienteActivo(cliente) {  
  const historicoDelCliente = baseDatos.buscar(cliente);  
  return historicoDelCliente.estaActivo();  
}
```

 **Volver arriba**

Los nombres de las funciones deberían decir lo que hacen

 **Mal:**

```
function añadirAFecha(fecha, mes) {  
  // ...  
}  
  
const fecha = new Date();  
  
// Es difícil saber que se le está añadiendo a la fecha en este caso  
añadirAFecha(fecha, 1);
```

 **Bien:**


```
function añadirMesAFecha(mes, fecha) {  
  // ...  
}  
  
const fecha = new Date();  
añadirMesAFecha(1, fecha);
```

 [Volver arriba](#)

Las funciones deberían ser únicamente de un nivel de abstracción

Cuando tienes más de un nivel de abstracción, tu función normalmente está haciendo demasiado. Separarla en funciones más pequeñas te ayudará a poder reutilizar código y te facilitará el *testear* éstas.

 **Mal:**

```
function analizarMejorAlternativaJavascript(codigo) {  
  const EXPRESIONES_REGULARES = [  
    // ...  
  ];  
  
  const declaraciones = codigo.split(" ");  
  const tokens = [];  
  EXPRESIONES_REGULARES.forEach(EXPRESION_REGULAR => {  
    declaraciones.forEach(declaracion => {  
      // ...  
    });  
  });  
  
  const ast = [];  
  tokens.forEach(token => {  
    // lex...  
  });  
  
  ast.forEach(nodo => {  
    // parse...  
  });  
}
```

 **Bien:**

```
function analizarMejorAlternativaJavascript(codigo) {  
  const tokens = tokenize(codigo);  
  const ast = lexer(tokens);  
  ast.forEach(nodo => {  
    // parse...  
  });  
}
```

```
}

function tokenize(codigo) {
  const EXPRESIONES_REGULARES = [
    // ...
  ];

  const declaraciones = codigo.split(" ");
  const tokens = [];
  EXPRESIONES_REGULARES.forEach(EXPRESION_REGULAR => {
    declaraciones.forEach(declaracion => {
      tokens.push(/* ... */);
    });
  });

  return tokens;
}

function lexer(tokens) {
  const ast = [];
  tokens.forEach(token => {
    ast.push(/* ... */);
  });

  return ast;
}
```

[↑ Volver arriba](#)

Elimina código duplicado

Haz todo lo posible para evitar duplicación de código. Duplicar código es malo porque significa que para editar un comportamiento... tendrás que modificarlo en más de un sitio. ¿Y no queremos trabajar de más, verdad?

Como caso práctico: Imagina que tienes un restaurante. Llevas el registro del inventario: Todos tus tomates, cebollas, ajos, especias, etc... Si tuvieras más de una lista que tuvieras que actualizar cada vez que sirves un tomate o usas una especie, sería más fácil de cometer errores, además de todo el tiempo perdido. Si solo tienes una, la posibilidad de cometer un error se reduce a ésta!

A menudo tienes código duplicado porque tienes dos o más cosas ligeramente diferentes, que tienen mucho en común, pero sus diferencias te obligan a tener ese código de más. Borrar la duplicación de código significa crear una abstracción que pueda manejar este conjunto de cosas diferentes con una sola función/módulo/clase.

Hacer que la abstracción sea correcta es fundamental y a veces bastante complejo. Es por eso que debes seguir los Principios SOLID establecidos en la sección *Clases*. Las malas abstracciones pueden ser peores que el código duplicado. ¡Así que ten cuidado! Dicho esto, si se puede hacer una buena abstracción, ¡Házla! Evita repetirte porque de lo contrario, como hemos comentado anteriormente, te verás editando en más de un lugar para modificar un comportamiento.

 **Mal:**

```
function mostrarListaDesarrolladores(desarrolladores) {
  desarrolladores.forEach(desarrollador => {
    const salarioEsperado = desarrollador.calcularSalarioEsperado();
    const experiencia = desarrollador.conseguirExperiencia();
    const enlaceGithub = desarrollador.conseguirEnlaceGithub();
    const datos = {
      salarioEsperado,
      experiencia,
      enlaceGithub
    };

    render(datos);
  });
}

function mostrarListaJefes(jefes) {
  jefes.forEach(jefe => {
    const salarioEsperado = desarrollador.calcularSalarioEsperado();
    const experiencia = desarrollador.conseguirExperiencia();
    const experienciaLaboral = jefe.conseguirProyectosMBA();
    const data = {
      salarioEsperado,
      experiencia,
      experienciaLaboral
    };

    render(data);
  });
}
```

 **Bien:**

```
function mostrarListaEmpleados(empleados) {
  empleados.forEach(empleado => {
    const salarioEsperado = empleado.calcularSalarioEsperado();
    const experiencia = empleado.conseguirExperiencia();

    const datos = {
      salarioEsperado,
      experiencia
    };
  });
}
```

```

    };

    switch (empleado.tipo) {
      case "jefe":
        datos.portafolio = empleado.conseguirProyectosMBA();
        break;
      case "desarrollador":
        datos.enlaceGithub = empleado.conseguirEnlaceGithub();
        break;
    }

    render(datos);
  });
}

```

 [Volver arriba](#)

Asigna objetos por defecto con Object.assign

 Mal:

```

const configuracionMenu = {
  titulo: null,
  contenido: "Bar",
  textoBoton: null,
  cancelable: true
};

function crearMenu(config) {
  config.titulo = config.titulo || "Foo";
  config.contenido = config.contenido || "Bar";
  config.textoBoton = config.textoBoton || "Baz";
  config.cancelable =
    config.cancelable !== undefined ? config.cancelable : true;
}

crearMenu(configuracionMenu);

```

 Bien:

```

const configuracionMenu = {
  titulo: "Order",
  // El usuario no incluyó la clave 'contenido'
  textoBoton: "Send",
  cancelable: true
};

function crearMenu(configuracion) {
  configuracion = Object.assign(

```

```

    {
      titulo: "Foo",
      contenido: "Bar",
      textoBoton: "Baz",
      cancelable: true
    },
    configuracion
  );

  // configuracion ahora es igual a: {titulo: "Order", contenido: "Bar", textoBoton:
  // ...
}

crearMenu(configuracionMenu);

```

[↑ Volver arriba](#)

No utilices banderas o flags

Las banderas o *flags* te indican de que esa función hace más de una cosa. Ya que como vamos repitiendo, nuestras funciones solo deberían hacer una cosa, separa esa lógica que es diferenciada por la bandera o *flag* en una nueva función.

 Mal:

```

function crearFichero(nombre, temporal) {
  if (temporal) {
    fs.create(`./temporal/${nombre}`);
  } else {
    fs.create(nombre);
  }
}

```

 Bien:

```

function crearFichero(nombre) {
  fs.create(nombre);
}

function crearFicheroTemporal(nombre) {
  crearFichero(`./temporal/${nombre}`);
}

```

[↑ Volver arriba](#)

Evita los efectos secundarios (parte 1)

Una función produce un efecto adverso/colateral si hace otra cosa que recibir un parámetro de entrada y retornar otro valor o valores. Un efecto adverso puede ser escribir un fichero, modificar una variable global o accidentalmente enviar todo tu dinero a un desconocido.

Ahora bien, a veces necesitamos efectos adversos en nuestros programas. Como en el ejemplo anterior, quizás necesitas escribir en un fichero. Así pues, lo que queremos es centralizar donde se hace esta acción. No queremos que esta lógica la tengamos que escribir en cada una de las funciones o clases que van a utilizarla. Para eso, la encapsularemos en un servicio que haga eso. Sólo eso.

El objetivo principal es evitar errores comunes como compartir el estado entre objetos sin ninguna estructura, usando tipos de datos mutables que pueden ser escritos por cualquier cosa y no centralizar donde se producen sus efectos secundarios. Si puedes hacer esto, serás más feliz que la gran mayoría de otros programadores.



Mal:

```
// Variable Global referenciada por la siguiente función
// Si tuviéramos otra función que usara ese nombre, podría ser un array y lo estaríamos
// If we had another function that used this name, now it'd be an array and it could
let nombre = 'Ryan McDermott';

function separarEnNombreYApellido() {
  nombre = nombre.split(' ');
}

separarEnNombreYApellido();

console.log(nombre); // ['Ryan', 'McDermott'];
```



Bien:

```
function separarEnNombreYApellido() {
  return nombre.split(' ');
}

const nombre = 'Ryan McDermott';
const nuevoNombre = separarEnNombreYApellido();

console.log(nombre); // 'Ryan McDermott';
console.log(nuevoNombre); // ['Ryan', 'McDermott'];
```



[Volver arriba](#)

Evita los efectos secundarios (parte 2)

En JavaScript, los primitivos se pasan por valor y los objetos / arrays se pasan por referencia. En el caso de objetos y arrays, si su función hace un cambio como por ejemplo, añadiendo un elemento al array que representa el carrito de la compra, entonces cualquier otra función que use ese array `carrito` se verá afectada por esta modificación. Eso puede ser genial, sin embargo, también puede ser malo. Imaginemos una mala situación:

El usuario hace clic en el botón "Comprar", que llama a una función de "compra" que genera una petición de red y envía el array `carrito` al servidor. Dada una mala conexión de red, la función `comprar` tiene que seguir reintentando la solicitud. Ahora, ¿Qué pasa si mientras tanto el usuario hace clic accidentalmente en el botón "Agregar al carrito" en un elemento que realmente no quiere, antes de que comience la solicitud de red? Si esto sucede y la solicitud de red comienza, entonces esa función de compra enviará el artículo agregado accidentalmente porque tiene una referencia al objeto dado que la función `añadirObjetoAlCarrito` modificó el `carrito` agregando un elemento que no deseado.

Una buena solución para `añadirObjetoAlCarrito` podría ser clonar el `carrito`, editarlo, y retornar la copia. Esto nos asegura que ninguna otra función tiene referencia al objeto con los campos modificados. Así pues, ninguna otra función se verá afectada por nuestros cambios.

Dos advertencias que mencionar para este enfoque:

1. Puede haber casos en los que realmente desee modificar el objeto de entrada, pero cuando adopte esta práctica de programación encontrará que esos casos son bastante raros ¡La mayoría de las cosas se pueden refactorizar para que no tengan efectos secundarios!
2. Clonar objetos grandes puede ser muy costosa en términos de rendimiento. Por suerte, en la práctica, esto no es un gran problema dado que hay [buenas librerías](#) que permiten este tipo de enfoque de programación. Es rápido y no requiere tanta memoria como te costaría a ti clonar manualmente los arrays y los objetos.



Mal:

```
const añadirObjetoAlCarrito = (carrito, objeto) => {  
  carrito.push({ objeto, fecha: Date.now() });  
};
```



Bien:

```
const añadirObjetoAlCarrito = (carrito, objeto) => {  
  return [...carrito, { objeto, fecha: Date.now() }];  
};
```



[Volver arriba](#)

No escribas en variables globales

La contaminación global es una mala práctica en JavaScript porque podría chocar con otra librería y usuarios usuarios de tu API no serían conscientes de ello hasta que tuviesen un error en producción. Pensemos en un ejemplo: ¿Qué pasaría si quisieras extender los arrays de Javascript para tener un método `diff` que pudiera enseñar la diferencia entre dos arrays? Podrías escribir tu nueva función en el `Array.prototype`, pero podría chocar con otra librería que intentó hacer lo mismo. ¿Qué pasa si esa otra librería estaba usando `diff` para encontrar la diferencia entre los elementos primero y último de una matriz? Tendríamos problemas... Por eso, sería mucho mejor usar las clases ES2015 / ES6 y simplemente extender el `Array` global.

 Mal:

```
Array.prototype.diff = function diff(matrizDeComparación) {  
  const hash = new Set(matrizDeComparación);  
  return this.filter(elemento => !hash.has(elemento));  
};
```

 Bien:

```
class SuperArray extends Array {  
  diff(matrizDeComparación) {  
    const hash = new Set(matrizDeComparación);  
    return this.filter(elemento => !hash.has(elemento));  
  }  
}
```

 [Volver arriba](#)

Da prioridad a la programación funcional en vez de la programación imperativa

Javascript no es un language funcional en la misma medida que lo es Haskell, pero tiene aspectos que lo favorecen. Los lenguajes funcionales pueden ser más fáciles y limpios de *testear*. Favorece este estilo de programación siempre que puedas.

 Mal:

```
const datosSalidaProgramadores = [  
  {  
    nombre: "Uncle Bobby",  
    lineasDeCodigo: 500  
  },  
  {  
    nombre: "Suzie Q",  
    lineasDeCodigo: 500  
  }  
];
```



```
    liniasDeCodigo: 1500
  },
  {
    nombre: "Jimmy Gosling",
    liniasDeCodigo: 150
  },
  {
    nombre: "Gracie Hopper",
    liniasDeCodigo: 1000
  }
];

let salidaFinal = 0;

for (let i = 0; i < datosSalidaProgramadores.length; i++) {
  salidaFinal += datosSalidaProgramadores[i].liniasDeCodigo;
}
```



Bien:

```
const datosSalidaProgramadores = [
  {
    nombre: "Uncle Bobby",
    liniasDeCodigo: 500
  },
  {
    nombre: "Suzie Q",
    liniasDeCodigo: 1500
  },
  {
    nombre: "Jimmy Gosling",
    liniasDeCodigo: 150
  },
  {
    nombre: "Gracie Hopper",
    liniasDeCodigo: 1000
  }
];

const salidaFinal = datosSalidaProgramadores
  .map(salida => salida.linesOfCode)
  .reduce((totalLinias, linias) => totalLinias + linias);
```



[Volver arriba](#)

Encapsula los condicionales



Mal:

```
if (fsm.state === "cogiendoDatos" && estaVacio(listaNodos)) {  
  // ...  
}
```



Bien:

```
function deberiaMostrarSpinner(fsm, listaNodos) {  
  return fsm.state === "cogiendoDatos" && estaVacio(listaNodos);  
}  
  
if (deberiaMostrarSpinner(fsmInstance, listNodeInstance)) {  
  // ...  
}
```



Volver arriba

Evita condicionales negativos



Mal:

```
function noEstaElNodoPresente(node) {  
  // ...  
}  
  
if (!noEstaElNodoPresente(node)) {  
  // ...  
}
```



Bien:

```
function estaElNodoPresente(node) {  
  // ...  
}  
  
if (estaElNodoPresente(node)) {  
  // ...  
}
```



Volver arriba

Evita condicionales

Esto parece una tarea imposible. Al escuchar esto por primera vez, la mayoría de la gente dice "¿*como voy a ser capaz de hacer cosas sin un if* "? La respuesta a eso, es que deberías usar polimorfismo para conseguir lo mismo en la gran mayoría de los casos. La segunda pregunta que normalmente la gente hace es, *¿Bueno está bien pero para que voy a querer hacerlo?* La respuesta es uno de los conceptos previos que hemos visto de *Código limpio*: Una función debería hacer únicamente una cosa. Cuando tienes una función o clase que posee un `if`, le estás diciendo al usuario que tu función está haciendo más de una cosa. Recuerda, tan sólo una cosa.

 **Mal:**

```
class Avion {
  // ...
  obtenerAlturaDeVuelo() {
    switch (this.tipo) {
      case "777":
        return this.cogerAlturaMaxima() - this.conseguirNumeroPasajeros();
      case "Air Force One":
        return this.cogerAlturaMaxima();
      case "Cessna":
        return this.cogerAlturaMaxima() - this.getFuelExpenditure();
    }
  }
}
```

 **Bien:**

```
class Avion {
  // ...
}

class Boeing777 extends Avion {
  // ...
  obtenerAlturaDeVuelo() {
    return this.cogerAlturaMaxima() - this.conseguirNumeroPasajeros();
  }
}

class AirForceOne extends Avion {
  // ...
  obtenerAlturaDeVuelo() {
    return this.cogerAlturaMaxima();
  }
}

class Cessna extends Avion {
  // ...
  obtenerAlturaDeVuelo() {
    return this.cogerAlturaMaxima() - this.getFuelExpenditure();
  }
}
```

```
}  
}
```

[!\[\]\(dfbd6b3763a6d1d9afaa974f64e2e4b5_img.jpg\) Volver arriba](#)

Evita el control de tipos (parte 1)

Javascript es un lenguaje no tipado. Esto significa que las funciones pueden recibir cualquier tipo como argumento. A veces, nos aprovechamos de eso... y es por eso, que se vuelve muy tentador el controlar los tipos de los argumentos de la función. Hay algunas soluciones para evitar esto. La primera, son APIs consistentes. Por API se entiende de que manera nos comunicamos con ese módulo/función.

 Mal:

```
function viajarATexas(vehiculo) {  
  if (vehiculo instanceof Bicicleta) {  
    vehiculo.pedalear(this.ubicacionActual, new Localizacion("texas"));  
  } else if (vehiculo instanceof Car) {  
    vehiculo.conducir(this.ubicacionActual, new Localizacion("texas"));  
  }  
}
```

 Bien:

```
function viajarATexas(vehiculo) {  
  vehiculo.mover(this.ubicacionActual, new Localizacion("texas"));  
}
```

[!\[\]\(dd161862f9164df98f62b726e9846241_img.jpg\) Volver arriba](#)

Evita control de tipos (parte 2)

Si estás trabajando con los tipos primitivos como son las cadenas o enteros, y no puedes usar polimorfismo pero aún ves la necesidad del control de tipos, deberías considerar Typescript. Es una excelente alternativa al Javascript convencional que nos aporta control de tipos de manera estática entre otras muchas cosas. El problema de controlar manualmente el tipado en Javascript es que para hacerlo bien, necesitamos añadir mucho código a bajo nivel que afecta a la legibilidad del código. Mantén tu código Javascript limpio, escribe tests y intenta tener revisiones de código. Si no, intenta cubrir el máximo de cosas con Typescript que como ya hemos dicho, es una muy buena alternativa.

 Mal:

```
function combina(valor1, valor2) {  
  if (  
    (typeof valor1 === "number" && typeof valor2 === "number") ||  
    (typeof valor1 === "string" && typeof valor2 === "string")  
  ) {  
    return valor1 + valor2;  
  }  
  
  throw new Error("Debería ser una cadena o número");  
}
```

**Bien:**

```
function combina(valor1, valor2) {  
  return valor1 + valor2;  
}
```

**Volver arriba**

No optimizes al máximo

Los navegadores modernos hacen mucha optimización por detrás en tiempo de ejecución. Muchas veces, al intentar optimizar tu código... estás perdiendo el tiempo. [Esta es una buena documentación](#) para ver donde falta optimización. Pon el foco en éstas hasta que estén arregladas/hechas si es que se pueden.

**Mal:**

```
// En los navegadores antiguos, cada iteración en la que `list.length` no esté cachea  
// podría ser costosa por el recálculo de este valor. En los modernos, ya está optimi  
for (let i = 0, tamaño = lista.length; i < tamaño; i++) {  
  // ...  
}
```

**Bien:**

```
for (let i = 0; i < lista.length; i++) {  
  // ...  
}
```

**Volver arriba**

Borra código inútil

El código inútil es tan malo como la duplicación. No hay razón alguna para mantenerlo en tu código. Si no está siendo usado por nadie, ¡Bórralo! Siempre estará disponible en sistema de versiones para el caso que lo necesites.

 **Mal:**

```
function antiguoModuloDePeticiones(url) {  
  // ...  
}  
  
function nuevoModuloDePeticiones(url) {  
  // ...  
}  
  
const petition = nuevoModuloDePeticiones;  
calculadorDeInventario("manzanas", petition, "www.inventory-awesome.io");
```

 **Bien:**

```
function nuevoModuloDePeticiones(url) {  
  // ...  
}  
  
const petition = nuevoModuloDePeticiones;  
calculadorDeInventario("manzanas", petition, "www.inventory-awesome.io");
```

 [Volver arriba](#)

Objetos y estructuras de datos

Utiliza setters y getters

Usar `getters` y `setters` para acceder a la información del objeto está mejor que simplemente accediendo a esa propiedad del objeto. ¿Por qué?

- Si quieres modificar una propiedad de un objeto, no tienes que ir mirando si existe o no existe para seguir mirando a niveles más profundos del objeto.
- Encapsula la representación interna (en caso de tener que comprobar cosas, mirar en varios sitios...)
- Es sencillo añadir mensajes y manejos de error cuando hacemos `get` y `set`
- Te permite poder hacer lazy load en caso de que los datos se recojan de una Base de Datos (bddd)

 **Mal:**

```
function crearCuentaBancaria() {  
  // ...  
  
  return {  
    balance: 0  
    // ...  
  };  
}  
  
const cuenta = crearCuentaBancaria();  
cuenta.balance = 100;
```



Bien:

```
function crearCuentaBancaria() {  
  // Esta es privada  
  let balance = 0;  
  
  // Un "getter", hecho público a través del objeto que retornamos abajo  
  function cogerBalance() {  
    return balance;  
  }  
  
  // Un "setter", hecho público a través del objeto que retornamos abajo  
  function introducirBalance(cantidad) {  
    // ... validamos antes de hacer un balance  
    balance = cantidad;  
  }  
  
  return {  
    // ...  
    cogerBalance,  
    introducirBalance  
  };  
}  
  
const cuenta = crearCuentaBancaria();  
cuenta.introducirBalance(100);
```



Volver arriba

Hacer que los objetos tengan atributos/métodos privados

Esto se puede hacer mediante `clojures` (de ES5 en adelante).



Mal:

```
const Empleado = function(nombre) {
  this.nombre = nombre;
};

Empleado.prototype.cogerNombre = function cogerNombre() {
  return this.nombre;
};

const empleado = new Empleado("John Doe");
console.log(`Nombre del empleado: ${empleado.cogerNombre()}`); // Nombre del empleado
delete empleado.nombre;
console.log(`Nombre del empleado: ${empleado.cogerNombre()}`); // Nombre del empleado
```

**Bien:**

```
function crearEmpleado(name) {
  return {
    cogerNombre() {
      return name;
    }
  };
}

const empleado = crearEmpleado("John Doe");
console.log(`Nombre del empleado: ${empleado.cogerNombre()}`); // Nombre del empleado
delete empleado.name;
console.log(`Nombre del empleado: ${empleado.cogerNombre()}`); // Nombre del empleado
```

[Volver arriba](#)

Clases

Prioriza las clases de ES2015/ES6 antes que las funciones planas de ES50

Es muy complicado de conseguir que un código sea entendible y fácil de leer con herencia de clases, construcción y metodos típicos de clases con las clases de ES5. Si necesitas herencia (y de seguro, que no la necesitas) entonces, dale prioridad a las clases ES2015/ES6. De todas las maneras, deberías preferir pequeñas funciones antes que ponerte a hacer clases. Solo cuando tengas un código largo o cuando veas necesaria la implementación de clases, añádelas.

**Mal:**


```

const Animal = function(edad) {
  if (!(this instanceof Animal)) {
    throw new Error("Inicializa Animal con `new`");
  }

  this.edad = edad;
};

Animal.prototype.mover = function mover() {};

const Mamifero = function(edad, furColor) {
  if (!(this instanceof Mamifero)) {
    throw new Error("Inicializa Mamifero con `new`");
  }

  Animal.call(this, edad);
  this.furColor = furColor;
};

Mamifero.prototype = Object.create(Animal.prototype);
Mamifero.prototype.constructor = Mamifero;
Mamifero.prototype.aniversario = function aniversario() {};

const Humano = function(edad, furColor, idioma) {
  if (!(this instanceof Humano)) {
    throw new Error("Inicializa Humano con `new`");
  }

  Mamifero.call(this, edad, furColor);
  this.idioma = idioma;
};

Humano.prototype = Object.create(Mamifero.prototype);
Humano.prototype.constructor = Humano;
Humano.prototype.hablar = function hablar() {};

```



Bien:

```

class Animal {
  constructor(edad) {
    this.edad = edad;
  }

  mover() {
    /* ... */
  }
}

class Mamifero extends Animal {
  constructor(edad, furColor) {

```

```
    super(edad);
    this.furColor = furColor;
}

aniversario() {
    /* ... */
}

}

class Human extends Mamifero {
    constructor(edad, furColor, idioma) {
        super(edad, furColor);
        this.idioma = idioma;
    }

    hablar() {
        /* ... */
    }
}
```

 [Volver arriba](#)

Utiliza el anidación de funciones

Este es un patrón útil en Javascript y verás que muchas librerías como jQuery o Lodash lo usan. Permite que tu código sea expresivo y menos verboso. Por esa razón, utiliza las funciones anidadas y date cuenta de que tan limpio estará tu código. En las funciones de tu clase, sencillamente retorna `this` al final de cada una y con eso, tienes todo lo necesario pra poder anidar las llamadas a las funciones.

 **Mal:**

```
class Coche {
    constructor(marca, modelo, color) {
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
    }

    introducirMarca(marca) {
        this.marca = marca;
    }

    introducirModelo(modelo) {
        this.modelo = modelo;
    }

    introducirColor(color) {
        this.color = color;
    }
}
```

```
    guardar() {  
        console.log(this.marca, this.modelo, this.color);  
    }  
}  
  
const coche = new Coche("Ford", "F-150", "rojo");  
coche.introducirColor("rosa");  
coche.guardar();
```



Bien:

```
class Coche {  
    constructor(marca, modelo, color) {  
        this.marca = marca;  
        this.modelo = modelo;  
        this.color = color;  
    }  
  
    introducirMarca(marca) {  
        this.marca = marca;  
        // NOTE: Retornamos this para poder anidas funciones  
        return this;  
    }  
  
    introducirModelo(modelo) {  
        this.modelo = modelo;  
        // NOTE: Retornamos this para poder anidas funciones  
        return this;  
    }  
  
    introducirColor(color) {  
        this.color = color;  
        // NOTE: Retornamos this para poder anidas funciones  
        return this;  
    }  
  
    guardar() {  
        console.log(this.marca, this.modelo, this.color);  
        // NOTE: Retornamos this para poder anidas funciones  
        return this;  
    }  
}  
  
const coche = new Coche("Ford", "F-150", "rojo")  
    .introducirColor("rosa")  
    .guardar();
```



Volver arriba

Prioriza la composición en vez de la herencia

Como se citó en [Patrones de Diseño](#) por "the Gang of Four", deberías priorizar la composición en vez de la herencia siempre que puedas. Hay muy buenas razones para usar tanto la herencia como la composición. El problema principal es que nuestra mente siempre tiende a la herencia como primera opción, pero deberíamos de pensar qué tan bien nos encaja la composición en ese caso particular porque en muchas ocasiones es lo más acertado.

Te estarás preguntando entonces, *¿Cuándo debería yo usar la herencia?* Todo depende. Depende del problema que tengas entre mano, pero ya hay ocasiones particulares donde la herencia tiene más sentido que la composición:

1. Tu herencia representa una relación "es un/a" en vez de "tiene un/a" (Humano->Animal vs. Usuario->DetallesUsuario)
2. Puedes reutilizar código desde las clases base (Los humanos pueden moverse como animales)
3. Quieres hacer cambios generales a clases derivadas cambiando la clase base. (Cambiar el consumo de calorías a todos los animales mientras se mueven)

 **Mal:**

```
class Empleado {
  constructor(nombre, correoElectronico) {
    this.nombre = nombre;
    this.correoElectronico = correoElectronico;
  }

  // ...
}

// Bad because Employees "have" tax data. EmployeeTaxData is not a type of Empleado
class InformacionImpuestosEmpleado extends Empleado {
  constructor(ssn, salario) {
    super();
    this.ssn = ssn;
    this.salario = salario;
  }

  // ...
}
```

 **Bien:**

```
class InformacionImpuestosEmpleado {
  constructor(ssn, salario) {
```

```
    this.ssn = ssn;
    this.salarario = salario;
}

// ...
}

class Empleado {
  constructor(nombre, correoElectronico) {
    this.nombre = nombre;
    this.correoElectronico = correoElectronico;
  }

  introducirInformacionImpuestos(ssn, salario) {
    this.informacionImpuestos = new InformacionImpuestosEmpleado(ssn, salario);
  }
  // ...
}
```

 [Volver arriba](#)

SOLID

Principio de Responsabilidad Única (SRP)

Como se cita en *Código Limpio*, "No debería haber nunca más de un motivo para que una clase cambie". Es muy tentador acribillar a una clase con un montón de funcionalidad. El problema que tiene esto, es que tu clase no tendrá cohesión y tendrá bastantes motivos por los que cambiar. Es por eso que es importante reducir el número de veces que tendrás que modificar una clase. Y lo es, porque en caso de que tengamos una clase que haga más de una cosa y modifiquemos una de ellas, no podemos saber que efectos colaterales puede tener esta acción en las demás.

 **Mal:**

```
class OpcionesUsuario {
  constructor(usuario) {
    this.usuario = usuario;
  }

  changeSettings(opciones) {
    if (this.verificarCredenciales()) {
      // ...
    }
  }

  verificarCredenciales() {
    // ...
  }
}
```

```
}  
}
```

**Bien:**

```
class AuthenticationUsuario {  
  constructor(usuario) {  
    this.usuario = usuario;  
  }  
  
  verificarCredenciales() {  
    // ...  
  }  
}  
  
class UserSettings {  
  constructor(usuario) {  
    this.usuario = usuario;  
    this.autenticacion = new AuthenticationUsuario(usuario);  
  }  
  
  changeSettings(settings) {  
    if (this.autenticacion.verificarCredenciales()) {  
      // ...  
    }  
  }  
}
```

**Volver arriba**

Principio de abierto/cerrado (OCP)

Citado por Bertrand Meyer: *"Las entidades de software (clases, módulos, funciones, ...) deberían estar abiertas a extensión pero cerradas a modificación."* ¿Qué significa esto?

Básicamente significa que los usuarios deberían de ser capaces de añadir funcionalidad a la aplicación sin tener que tocar el código creado hasta ahora.

**Mal:**

```
class AdaptadorAjax extends Adaptador {  
  constructor() {  
    super();  
    this.name = "adaptadorAjax";  
  }  
}  
  
class AdaptadorNodos extends Adaptador {  
  constructor() {
```

```

    super();
    this.nombre = "adaptadorNodos";
  }
}

class {
  constructor(adapter) {
    this.adapter = adapter;
  }

  fetch(url) {
    if (this.adapter.nombre === "adaptadorAjax") {
      return hacerLlamadaAjax(url).then(respuesta => {
        // transformar la respuesta y devolverla
      });
    } else if (this.adapter.nombre === "adaptadorHttpNodos") {
      return hacerLlamadaHttp(url).then(respuesta => {
        // transformar la respuesta y devolverla
      });
    }
  }
}

function hacerLlamadaAjax(url) {
  // request and return promise
}

function hacerLlamadaHttp(url) {
  // request and return promise
}

```



Bien:

```

class AdaptadorAjax extends Adapter {
  constructor() {
    super();
    this.nombre = "adaptadorAjax";
  }

  pedir(url) {
    // Pedir y devolver la promesa
  }
}

class AdaptadorNodos extends Adapter {
  constructor() {
    super();
    this.nombre = "adaptadorNodos";
  }

  pedir(url) {

```

```
// Pedir y devolver la promesa
}
}

class EjecutadorPeticionesHttp {
  constructor(adaptador) {
    this.adaptador = adaptador;
  }

  fetch(url) {
    return this.adaptador.pedir(url).then(respuesta => {
      // Transformar y devolver la respuesta
    });
  }
}
```

 [Volver arriba](#)

Principio de sustitución de Liskov (LSP)

Este es un término que asusta para lo sencillo que es. Estrictamente se define como "Si S es un subtipo de T, entonces los objetos del tipo T deberían poderse substituir por objetos del tipo S".

Un ejemplo práctico vien a ser si tenemos una *clase padre* y una *clase hija*, entonces ambas han de poderse substituir la una por la otra y viceversa sin recibir ningún tipo de error o datos erróneos. Un caso práctico es el del cuadrado y el rectángulo. Geométricamente, un cuadrado es un rectángulo, pero si lo creamos con una relación "es un" a través de herencia, empezamos a tener problemas...



Mal:

```
class Rectangulo {
  constructor() {
    this.anchura = 0;
    this.altura = 0;
  }

  introducirColor(color) {
    // ...
  }

  render(area) {
    // ...
  }

  introducirAnchura(anchura) {
    this.anchura = anchura;
  }
}
```



```

    introducirAltura(altura) {
        this.altura = altura;
    }

    conseguirArea() {
        return this.anchura * this.altura;
    }
}

class Cuadrado extends Rectangulo {
    introducirAnchura(anchura) {
        this.anchura = anchura;
        this.altura = anchura;
    }

    introducirAltura(altura) {
        this.width = altura;
        this.altura = altura;
    }
}

function renderizaRectangulosLargos(rectangulos) {
    rectangulos.forEach(rectangulo => {
        rectangulo.introducirAnchura(4);
        rectangulo.introducirAltura(5);
        const area = rectangulo.conseguirArea(); // MAL: Para el cuadrado devuelve 25 y d
        rectangulo.render(area);
    });
}

const rectangulos = [new Rectangulo(), new Rectangulo(), new Cuadrado()];
renderizaRectangulosLargos(rectangulos);

```



Bien:

```

class Forma {
    introducirColor(color) {
        // ...
    }

    render(area) {
        // ...
    }
}

class Rectangulo extends Forma {
    constructor(width, height) {
        super();
        this.anchura = anchura;
    }
}

```

```
    this.altura = altura;
  }

  conseguirArea() {
    return this.anchura * this.altura;
  }
}

class Cuadrado extends Forma {
  constructor(distancia) {
    super();
    this.distancia = distancia;
  }

  conseguirArea() {
    return this.distancia * this.distancia;
  }
}

function renderizaRectangulosLargos(shapes) {
  shapes.forEach(shape => {
    const area = shape.conseguirArea();
    shape.render(area);
  });
}

const shapes = [new Rectangulo(4, 5), new Rectangulo(4, 5), new Cuadrado(5)];
renderizaRectangulosLargos(shapes);
```

 [Volver arriba](#)

Principio de Segregacion de Interfaces (ISP)

Javascript no dispone de interfaces así que no podemos aplicar el principio como tal. De todas maneras, es importante conceptualmente hablando aunque no tengamos tipados como tal, pues eso resulta haciendo un código mantenible igualmente.

ISP dice que "los servicios no deberían estar forzados a depender de interfaces que realmente no usan".

Un buen ejemplo en javascript sería las típicas clases que requieren de un enormes objetos de configuración. No hacer que los servicios requieran de grandes cantidades de opciones es beneficioso, porque la gran mayoría del tiempo, no necesitarán esa configuración. Hacerlos opcionales ayuda a no tener el problema de "Interfaz gorda", en inglés conocido como "fat interface".



Mal:

```

class DOMTraverser {
  constructor(configuraciones) {
    this.configuraciones = configuraciones;
    this.setup();
  }

  preparar() {
    this.nodoRaiz = this.configuraciones.nodoRaiz;
    this.ModuloAnimacion.preparar();
  }

  atravesar() {
    // ...
  }
}

const $ = new DOMTraverser({
  nodoRaiz: document.getElementsByTagName("body"),
  moduloAnimacion() {} // Most of the time, we won't need to animate when traversing.
  // ...
});

```



Bien:

```

class DOMTraverser {
  constructor(configuraciones) {
    this.configuraciones = configuraciones;
    this.opciones = configuraciones.opciones;
    this.preparar();
  }

  preparar() {
    this.nodoRaiz = this.configuraciones.nodoRaiz;
    this.prepararOpciones();
  }

  prepararOpciones() {
    if (this.opciones.moduloAnimacion) {
      // ...
    }
  }

  atravesar() {
    // ...
  }
}

const $ = new DOMTraverser({
  nodoRaiz: document.getElementsByTagName("body"),

```

```
    opciones: {  
      moduloAnimacion() {}  
    }  
  });
```

 [Volver arriba](#)

Principio de Inversión de Dependencias (DIP)

Por favor, no confundir con Inyección de Dependencias. Mucha gente se piensa que la "D" de SOLID es de Inyección de Dependencias (*Dependency Inection, DI*).

Este principio nos dice dos cosas básicamente:

1. Módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
2. Las abstracciones no deberían depender de detalles si no que, los detalles deberían depender de abstracciones.

Esto puede ser algo complejo al principio, pero si has trabajado con AngularJS, has visto de manera indirecta esto con la Inyección de Dependencias. Como comentaba anteriormente, aunque no son lo mismo, van de la mano. La Inversión de Dependencias es posible gracias a la Inyección de Dependencias. *DI* hace posible que los módulos de alto nivel dependan de abstracciones y no de detalles.

El mayor de los beneficioses la reducción del acoplamiento entre módulos. Cuánto mayor acoplamiento, mayor dificultad en refactorización.

Como hemos comentado antes, Javascript no tiene interfaces así que los contratos son un poco... así así. Están en nuestra cabeza y eso debemos tenerlo en cuenta. Mucha gente usa javascript docs, anotaciones en comentarios justo encima de los módulos y algunas cosas más. Vamos a ver un ejemplo con `RastreadorDeInventario`.

 **Mal:**

```
class SolicitadorDeInventario {  
  constructor() {  
    this.REQ_METHODS = ["HTTP"];  
  }  
  
  pedirArticulo(articulo) {  
    // ...  
  }  
}
```

```
class RastreadorDeInventario {  
  constructor(articulos) {  
    this.articulos = articulos;  
  }  
}
```

```

    // MAL: Hemos creado una dependencia de una concreción que va atada a una impleme
    // Deberíamos tener pedirArticulos dependiendo únicamente de un método: 'solicit
    this.solicitador = new SolicitadorDeInventario();
  }

  pedirArticulos() {
    this.articulos.forEach(articulo => {
      this.solicitador.pedirArticulo(articulo);
    });
  }
}

const rastreadorDeInventario = new RastreadorDeInventario([
  "manzanas",
  "platanos"
]);
rastreadorDeInventario.pedirArticulos();

```



Bien:

```

class RastreadorDeInventario {
  constructor(articulos, solicitador) {
    this.articulos = articulos;
    this.solicitador = solicitador;
  }

  pedirArticulos() {
    this.articulos.forEach(articulo => {
      this.solicitador.pedirArticulo(articulo);
    });
  }
}

class SolicitadorDeInventarioV1 {
  constructor() {
    this.REQ_METHODS = ["HTTP"];
  }

  pedirArticulo(articulo) {
    // ...
  }
}

class SolicitadorDeInventarioV2 {
  constructor() {
    this.REQ_METHODS = ["WS"];
  }

  pedirArticulo(articulo) {

```

```
// ...  
}  
}  
  
// By constructing our dependencies externally and injecting them, we can easily  
// substitute our request module for a fancy new one that uses WebSockets.  
  
// Construyendo nuestras dependencias desde fuera e inyectandolas, podríamos  
// substituir nuestro Módulo solicitador por uno con websockets o lo que sea  
const rastreadorDeInventario = new RastreadorDeInventario(  
  ["manzanas", "platanos"],  
  new SolicitadorDeInventarioV2()  
);  
rastreadorDeInventario.pedirArticulos();
```

 [Volver arriba](#)

Testing

El testing es más importante que la entrega. Si no tienes test o tienes muchos que no soy de gran ayuda, cada vez que quieras entregar valor no estarás seguro de que eso funciona debidamente y que nada falla. Puedes decidir con el equipo cuál es el porcentaje al que queréis ceñiros pero, la única manera de tener confianza total de que nada falla, es teniendo 100% de cobertura de test. Para esto, necesitarás tener una gran herramienta para poder testear pero también una que te calcule adecuadamente [el porcentaje cubierto](#).

No hay excusas para no escribir tests. Hay [un montón de frameworks de JS](#) entre los que podréis tu y tu equipo decidir. Una vez hayáis elegido el framework, para cada nueva funcionalidad que se quiera añadir a la plataforma, escribir tests. Si prefieres hacer *Test-Driven Development* me parece bien, pero la idea principal de los test es dar confianza suficiente al programador para que pueda seguir entregando valor.

Sólo un concepto por test

 Mal:

```
import assert from "assert";  
  
describe("MakeMomentJSGreatAgain", () => {  
  it("maneja límites de las fechas", () => {  
    let fecha;  
  
    fecha = new MakeMomentJSGreatAgain("1/1/2015");  
    fecha.addDays(30);  
    assert.equal("1/31/2015", fecha);  
  
    fecha = new MakeMomentJSGreatAgain("2/1/2016");
```

```

    fecha.addDays(28);
    assert.equal("02/29/2016", fecha);

    fecha = new MakeMomentJSGreatAgain("2/1/2015");
    fecha.addDays(28);
    assert.equal("03/01/2015", fecha);
  });
});

```



Bien:

```

import assert from "assert";

describe("MakeMomentJSGreatAgain", () => {
  it("Maneja los meses con 30 días", () => {
    const fecha = new MakeMomentJSGreatAgain("1/1/2015");
    fecha.addDays(30);
    assert.equal("1/31/2015", fecha);
  });

  it("Maneja los años bisiestos", () => {
    const fecha = new MakeMomentJSGreatAgain("2/1/2016");
    fecha.addDays(28);
    assert.equal("02/29/2016", fecha);
  });

  it("Maneja los años NO bisiestos", () => {
    const fecha = new MakeMomentJSGreatAgain("2/1/2015");
    fecha.addDays(28);
    assert.equal("03/01/2015", fecha);
  });
});

```



[Volver arriba](#)

Concurrencia

Usa Promesas, no callbacks

Los callbacks son funciones que se pasan como parámetros a otras funciones para ser ejecutadas una vez esta función termina. Por ejemplo: Dada las funciones A y B, se dice que B es el callback de A si la función B es pasada como parámetro a la función A y esta, se ejecuta este callback una vez ha terminado

Los `callbacks` no son limpios ni en cuanto a legibilidad ni en cuanto a formato de texto (dado que provocan niveles de indentación). Con ES2015/ES6 las promesas son un tipo global. ¡Úsalas!

 Mal:

```
import { get } from "request";
import { writeFile } from "fs";

get(
  "https://en.wikipedia.org/wiki/Robert_Cecil_Martin",
  (requestErr, respuesta) => {
    if (requestErr) {
      console.error(requestErr);
    } else {
      writeFile("article.html", respuesta.body, writeErr => {
        if (writeErr) {
          console.error(writeErr);
        } else {
          console.log("File written");
        }
      });
    }
  }
);
```

 Bien:

```
import { get } from "request";
import { writeFile } from "fs";

get("https://en.wikipedia.org/wiki/Robert_Cecil_Martin")
  .then(respuesta => {
    return writeFile("article.html", respuesta);
  })
  .then(() => {
    console.log("File written");
  })
  .catch(err => {
    console.error(err);
  });
```

 [Volver arriba](#)

Async/Await is incluso más limpio que las Promesas

Las promesas son una elección más limpia que los callbacks pero ES2017/ES8 trae la funcionalidad de `async/await` que es incluso más limpio que las promesas. Todo lo que tienes que hacer es añadir el prefijo `async` a una función y entonces ya podemos usar esa función de manera imperativa sin ningún `.then()`. La palabra `await` la usarás para hacer que ese código asíncrono se comporte de "manera síncrona".

 **Mal:**

```
import { get } from "request-promise";
import { writeFile } from "fs-promise";

get("https://en.wikipedia.org/wiki/Robert_Cecil_Martin")
  .then(respuesrta => {
    return writeFile("article.html", respuesrta);
  })
  .then(() => {
    console.log("File written");
  })
  .catch(err => {
    console.error(err);
  });
```

 **Bien:**

```
import { get } from "request-promise";
import { writeFile } from "fs-promise";

async function conseguirArticulosDeCodigoLimpio() {
  try {
    const respuesrta = await get(
      "https://en.wikipedia.org/wiki/Robert_Cecil_Martin"
    );
    await writeFile("article.html", respuesrta);
    console.log("File written");
  } catch (err) {
    console.error(err);
  }
}
```

 **Volver arriba**

Manejo de errores

¡Lanzar errores está bien! Significa que en tiempo de ejecución se ha detectado que algo no estaba funcionando como debía y se ha parado la ejecución del trozo de código. Además se notifica siempre en la consola del navegador.

No ignores los errores capturados

No hacer nada con los errores capturados no te da la opción de anticiparte o arreglar dicho error. El printar el error por la consola del navegador no es una solución, pues la gran mayoría de veces nadie es consciente de eso y el error pasas desapercibido. Envuelve tu código con `try/catch` y es ahí donde tendrás que elaborar tu plan de reacción a posibles errores

 **Mal:**

```
try {  
  functionQueDeberiaLanzarError();  
} catch (error) {  
  console.log(error);  
}
```

 **Bien:**

```
try {  
  functionQueDeberiaLanzarError();  
} catch (error) {  
  // Una option (algo más molesta que el convencional console.log)  
  console.error(error);  
  // Otra opción:  
  notificarAlUsuarioDelError(error);  
  // Otra opción:  
  reportarElErrorAUnServicio(error);  
  // O hazlas todas!  
}
```

No ignores las promesas rechazadas

No ignores las promesas que han sido rechazadas por la misma razón que no deberías ignorar errores capturados en el `try/catch`.

 **Mal:**

```
cogerDatos()  
  .then(datos => {  
    functionQueDeberiaLanzarError(datos);  
  })  
  .catch(error => {  
    console.log(error);  
  });
```

 **Bien:**

```
cogerDatos()  
  .then(datos => {  
    functionQueDeberiaLanzarError(datos);  
  })  
  .catch(error => {  
    // Una option (algo más molesta que el convencional console.log)  
    console.error(error);  
    // Otra opción:  
    notificarAlUsuarioDelError(error);  
    // Otra opción:  
    reportarElErrorAUnServicio(error);  
    // O hazlas todas!  
  });
```

 [Volver arriba](#)

Formato

El formato del código es algo subjetivo. Como otras reglas aquí, no hay una regla que deberías seguir o una fórmula secreta. Lo que si que está claro es que no deberíamos discutir ni crear conflictos con nuestros compañeros de trabajo acerca de estas reglas. Hay unas cuantas [herramientas](#) que automatizan todas las reglas de formato de texto. ¡Ahorrarse tiempo en estas formateando el texto es un pasada!

Usa consistenemente la capitalización

Como ya hemos dicho, `javascript` es un language no tipado así pues, la capitalización de las variables importa, y mucho. Estas son reglas totalmente subjetivas así que como equipo, podéis elegir lo que más os guste/convenga. La cuestión es que independientemente de lo que decidáis, seáis consistentes.

 **Mal:**

```
const DIAS_POR_SEMANA = 7;  
const diasPorMes = 30;  
  
const canciones = ["Back In Black", "Stairway to Heaven", "Hey Jude"];  
const Artistas = ["ACDC", "Led Zeppelin", "The Beatles"];  
  
function borrarBaseDeDatos() {}  
function restablecer_baseDeDatos() {}  
  
class animal {}  
class Alpaca {}
```

 **Bien:**

```
const DIAS_POR_SEMANA = 7;
const DIAS_POR_MES = 30;

const CANCIONES = ["Back In Black", "Stairway to Heaven", "Hey Jude"];
const ARTISTAS = ["ACDC", "Led Zeppelin", "The Beatles"];

function borrarBaseDeDatos() {}
function restablecerBaseDeDatos() {}

class Animal {}
class Alpaca {}
```

 [Volver arriba](#)

Funciones que llaman y funciones que son llamadas, deberían estar cerca

Si una función llama a otra, haz que esta función que va a ser llamada esté lo más cerca posible de la función que la llama. Idealmente, sitúa siempre la función que va a ser llamada justo después de la función que la ejecuta. ¿El motivo? Pues normalmente acostumbramos a leer de arriba abajo y tampoco queremos tener que hacer *scroll* hasta abajo del todo del fichero para volver a subir.

 **Mal:**

```
class RevisionDeRendimiento {
  constructor(empleado) {
    this.empleado = empleado;
  }

  conseguirCompañeros() {
    return db.buscar(this.empleado, "compañeros");
  }

  conseguirJefe() {
    return db.buscar(this.empleado, "jefe");
  }

  conseguirOpinionDeLosCompañeros() {
    const compañeros = this.conseguirCompañeros();
    // ...
  }

  ejecutarRevision() {
    this.conseguirOpinionDeLosCompañeros();
    this.conseguirOpinionDelJefe();
    this.conseguirAutoRevision();
  }
}
```

```
    conseguirOpinionDelJefe() {
        const jefe = this.conseguirJefe();
    }

    conseguirAutoRevision() {
        // ...
    }
}

const review = new RevisionDeRendimiento(empleado);
review.executarRevision();
```



Bien:

```
class RevisionDeRendimiento {
    constructor(empleado) {
        this.empleado = empleado;
    }

    ejecutarRevision() {
        this.conseguirOpinionDeLosCompañeros();
        this.conseguirOpinionDelJefe();
        this.conseguirAutoRevision();
    }

    conseguirOpinionDeLosCompañeros() {
        const compañeros = this.conseguirCompañeros();
        // ...
    }

    conseguirCompañeros() {
        return db.buscar(this.empleado, "compañeros");
    }

    conseguirOpinionDelJefe() {
        const jefe = this.conseguirJefe();
    }

    conseguirJefe() {
        return db.buscar(this.empleado, "jefe");
    }

    conseguirAutoRevision() {
        // ...
    }
}

const review = new RevisionDeRendimiento(empleado);
review.executarRevision();
```

[↑ Volver arriba](#)

Comentarios

Comenta únicamente la lógica de negocio que es compleja

Los comentarios son una disculpa, no un requerimiento. Supuestamente se dice que un buen código debería comentarse por si mismo. Un código perfecto no está optimizado para la máquina sino que lo está para la mantenibilidad de éste por un compañero o futuro compañero. Para esto, ha de ser lo más semántico posible. El código ha de estar escrito para que niños pequeños lo entiendan.

 **Mal:**

```
function hashIt(datos) {  
  // El hash  
  let hash = 0;  
  
  // Tamaño del string  
  const tamaño = datos.length;  
  
  // Iteramos a través de cada carácter de los datos  
  for (let i = 0; i < tamaño; i++) {  
    // Coger código del carácter  
    const char = datos.charCodeAt(i);  
    // Crear el hash  
    hash = (hash << 5) - hash + char;  
    // Convertir a un entero de 32 bits  
    hash &= hash;  
  }  
}
```

 **Bien:**

```
function hashIt(datos) {  
  let hash = 0;  
  const tamaño = datos.length;  
  
  for (let i = 0; i < tamaño; i++) {  
    const caracter = datos.charCodeAt(i);  
    hash = (hash << 5) - hash + caracter;  
  
    // Convertir a un entero de 32 bits  
    hash &= hash;  
  }  
}
```

[↑ Volver arriba](#)

No dejes código comentado en tu repositorio

El control de versiones existe para algo. Si tu motivo o excusa por el que comentar un código es porque en breves o algún día lo vas a necesitar, eso no me sirve. Ese código que acabas de borrar consta en alguna de tus versiones de tu código fuente. Lo que deberías hacer entonces quizás, es usar `git tags`, poner el código de la tarea en el nombre del commit, etc... Hay muchos trucos para hacer eso!

 **Mal:**

```
hacerCosas();  
// hacerOtrasCosas();  
// hacerCosasAunMasRaras();  
// estoHaceMaravillas();
```

 **Bien:**

```
hacerCosas();
```

[↑ Volver arriba](#)

No hagas un diario de comentarios

Recuerda ¡Usa el control de versiones! No hay motivo alguno para tener código muerto, código comentado y aún menos, un diario o resumen de modificaciones en tus comentarios. Si quieres ver las modificaciones, usa `git log`, la herramienta `blame` o incluso el `history`.

 **Mal:**

```
/**  
 * 2016-12-20: `monads` borrados, no hay quien los entienda  
 * 2016-10-01: Código mejorado con 'monads'  
 * 2016-02-03: Borrado tipado  
 * 2015-03-14: Añadido tipado  
 */  
function combinar(a, b) {  
  return a + b;  
}
```

 **Bien:**

```
function combinar(a, b) {  
  return a + b;  
}
```

 [Volver arriba](#)

Evita los marcadores de secciones

Normalmente acostumbran a ser molestos. Deja que las variables y las funciones hagan su función con sus identaciones naturales y de esta manera, formateen el código correctamente .

 **Mal:**

```
////////////////////////////////////  
// Instanciación del Modelo Scope  
////////////////////////////////////  
$scope.modelo = {  
  menu: "foo",  
  nav: "bar"  
};  
  
////////////////////////////////////  
// Preparación de la acción  
////////////////////////////////////  
const acciones = function() {  
  // ...  
};
```

 **Bien:**

```
$scope.modelo = {  
  menu: "foo",  
  nav: "bar"  
};  
  
const acciones = function() {  
  // ...  
};
```

 [Volver arriba](#)

Traducciones

También esta disponible en otros idiomas

-  **Brazilian Portuguese:** [fesnt/clean-code-javascript](https://github.com/fesnt/clean-code-javascript)